

## Organización del Computador 2

### Primer parcial – 10/05/18

1 (40)	2 (40)	3 (20)	
40	30	20	90 (A)

#### Normas generales

- Numere las hojas entregadas. Complete en la primera hoja la cantidad total de hojas entregadas.
- Entregue esta hoja junto al examen, la misma **no** se incluye en la cantidad total de hojas entregadas.
- Está permitido tener los manuales y los apuntes con las listas de instrucciones en el examen. Está prohibido compartir manuales o apuntes entre alumnos durante el examen.
- Cada ejercicio debe realizarse en hojas separadas y numeradas. Debe identificarse cada hoja con nombre, apellido y LU.
- La devolución de los exámenes corregidos es personal. Los pedidos de revisión se realizarán por escrito, antes de retirar el examen corregido del aula.
- Los parciales tienen tres notas: I (Insuficiente): 0 a 59 pts, A- (Aprobado condicional): 60 a 64 pts y A (Aprobado): 65 a 100 pts. No se puede aprobar con A- ambos parciales. Los recuperatorios tienen dos notas: I: 0 a 64 pts y A: 65 a 100 pts.

### Ej. 1. (40 puntos)

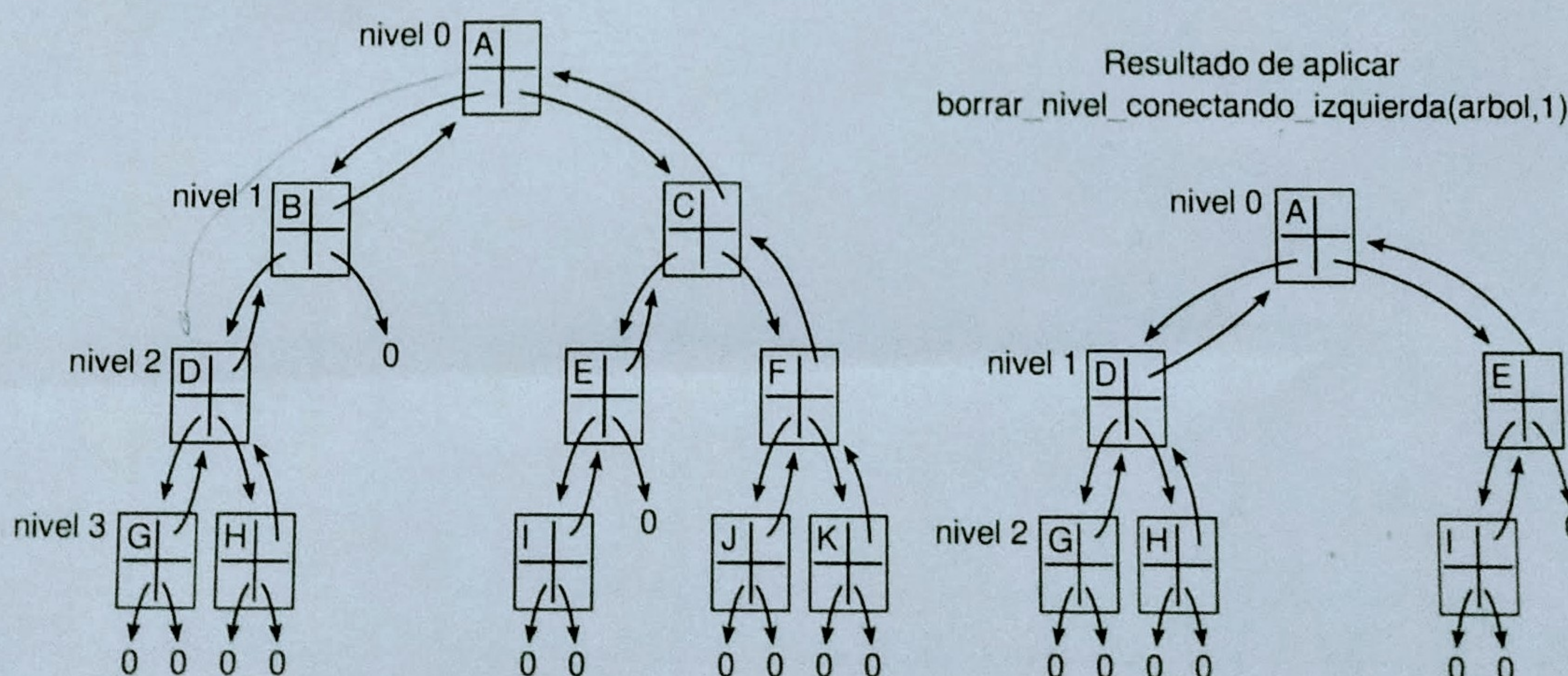
Sea un árbol binario doblemente enlazado que respeta la siguiente estructura:

```

struct nodo {
    struct nodo* derecho, 8  => 0
    struct nodo* izquierdo, 8
    struct nodo* padre, 8  16
    void* data, 8  24
    void (*borrar)(void*) 8  32
}
    
```

offsets  
Size 40 B

- (15p) a. Programar en ASM la función `ContarPorNivel` que dado un puntero a nodo y un número de nivel, cuenta la cantidad de nodos que hay en el nivel indicado. El resultado es retornado en un puntero a entero denominado `cantidad`. Su aridad es: `void contar_por_nivel(struct nodo* arbol, unsigned int nivel, unsigned int* cantidad)`.
- (25p) b. Programar en ASM la función `borrar_nivel_conectando_izquierda` que dado un doble puntero a nodo y un número de nivel, borra todos los nodos del nivel indicado, conectando al padre solamente los hijos izquierdos. El subárbol derecho debe ser eliminado. Su aridad es: `void borrar_nivel_conectando_izquierda(struct nodo** arbol, unsigned int nivel)`. Considerar que el puntero al primer nodo puede cambiar y debe ser retornado en el parámetro `arbol`.



Nota: Para borrar data se debe llamar a la función almacenada en el nodo en `borrar`. La misma tienen la misma aridad que la función `free`.



**Ej. 2. (40 puntos)**

Considerar un vector de 16 números enteros con signo de 24 bits almacenados en big-endian.

- 15 (25p) a. Construir una función en ASM utilizando SIMD que dado un puntero al vector de números mencionado, retorne la suma de los números del vector como un entero de 4 bytes.
- 15 (15p) b. Modificar la función anterior para que a los números pares los multiplique por  $\pi$ . En este caso el resultado debe ser retornado como *double*.

**Ej. 3. (20 puntos)**

Una nueva opción de compilación de gcc hace que luego de cualquier instrucción `call` se generen exactamente `k` bytes con *null*. Este lugar es utilizado por herramientas de *debugging* para almacenar información especial. El código resultante sería de la forma:

...|inst|inst|inst|call| k bytes |inst|inst|inst|...

El problema aparece cuando las funciones finalizan. Cuando se ejecute la instrucción `ret` se retornará a la dirección siguiente al `call`. En el código con la nueva opción de compilación, esto resultaría en la ejecución del área reservada de `k` bytes.

Para evitar este problema se pide construir una función que “arregle” la dirección de retorno. Esta función será ejecutada dentro de cada función llamada, en cualquier momento luego de construir el *stack-frame*.

*el compilador no hace padding en estz.*

- 6 (6p) a. Programar en ASM la función arregla\_retorno que modifica la dirección de retorno de una función para evitar ejecutar el área reservada. El valor `k` es un parámetro para `arreglar_retorno`.
- 7 (7p) b. Programar en ASM una función alternativa (`modificar_area_reservada`), que a diferencia de modificar la dirección de retorno, escribe instrucciones de *no-operacion* en el área reservada a fin de continuar con la correcta ejecución en caso de retornar sobre el área reservada. La instrucción NOP se codifica como `0x90` y ocupa 1 byte.
- 7 (7p) c. Decidir verdadero o falso y justificar: Desensamblar un archivo binario sin ningún tipo de información de debug, de código generado con la nueva opción de compilación, no presenta ningún problema adicional que desensamblar código que no utilice la mencionada opción.

**Nota:** `arreglar_retorno` y `modificar_area_reservada` son llamadas como cualquier otra función.



1) a) Para resolver este problema usaremos recursividad.

Basicamente la función contar por nivel es contar recursivo y devuelve el valor que este le pida. Veamos en pseudo código contar recursivo.

```
int CR(nodo * arbol, int nivel, int nivelActual)
{
    if (arbol == 0)
        return 0;

    if (nivelActual > nivel)
        return 0;

    if (nivelActual == nivel)
        return 1;

    int r1 = CR(arbol->derecho, nivel, nivelActual + 1);
    int r2 = CR(arbol->izquierda, nivel, nivelActual + 1);

    return r1 + r2;
}
```

Si el nodo está en el nivel buscado retornar 1. Si no retornar 0.  
Si le corresponde llamar a la recursión por sus hijos.

Section .data:

```
% define OFFSET_der 0
% " " OFFSET_izq 8
```

Section .text:

```
global contar_por_nivel; rdi = arbol
; esi = nivel
; edx = cantidad
```

```
push rbp
mov rbp, rsp
push rdx
sub rsp, 8
shl esi, 32 ; limpiar la parte alta.
shr esi, 32
```

```
xor rdx, rdx
```

Call contar\_recursivo

```
shl rax, 32
shr rax, 32
pop rdx
add rsp, 8
pop rdx
mov [rdi], rax
```

```
pop rbp
ret
```

```
shl rax, 32
shr rax, 32
pop rdx
add rsp, 8
pop rdx
mov [rdi], rax
ret
```



global Contar Recursivo 2

; rdi nodo Actual  
; esi nivel  
; edx nivel actual

push rbp ; 21  
mov rbp, rsp  
push rsi ; der  
push rdx ; 21  
push rdi ; der  
push r12 ; 21  
push r13 ; der  
push r14 ; 21  
push r15 ; der  
sub rsp, 8 ; 21  
cmp ~~rdi~~ <sup>rdi</sup>, 0  
je .fin

add rsp, 8  
pop r15  
pop r14  
pop r13  
pop r12  
pop rdi  
pop rdx  
pop rsi  
pop rbp  
ret

ahora que lo pienso,  
creo que no hacer Kaltura  
preservados.

shl rsi, 32  
shr rsi, 32  
shl rdx, 32  
shr rdx, 32

mov r12, rsi  
mov r13, rdx  
mov r14, rdi  
xor r15, r15  
~~xor rax, rax~~  
cmp rax, rsi  
jg .fin  
je .Suma Uno

mov rsi, r12  
inc r13  
mov rdx, r13  
mov rdi, [r14 + offset - der]  
call Contar Recursivo

mov r15, rax  
mov rdx, r13  
mov rsi, r12  
mov rdi, [r14 + offset - izq]  
call Contar Recursivo  
add r15, rax  
mov rax, r15  
jmp .fin

.Suma Uno  
inc rax

.fin



Borrar

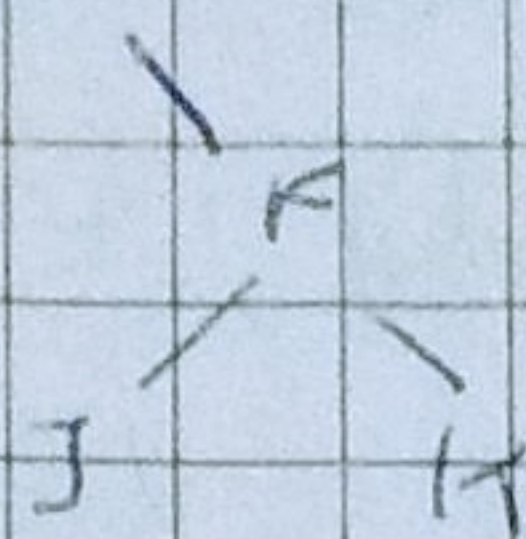
1) Si llamo nivel 3, tengo que borrar B, C

- Debo bajar al nivel adecuado.

- Borrar todos los hijos  
el subárbol derecho

- Recorrer para dejar libre el nodo a borrar

- Liberar el nodo. (primero dato).



Buscar Nivel (nodo x arbol, nivel, nivel Actual, bool Soy Derecho)

if (arbol != 0)

if (nivel Actual == nivel Actual)

Buscar Nivel (arbol → izq, nivel, nivel Actual + 1, false)

Buscar Nivel (arbol → der, nivel, nivel Actual + 1, true)

if (nivel == nivel Actual) // ESTE IF ES TRIVIAL, NO SE INCLUYE EN ASM

Borrar SubArbol (arbol → der)

if (nivel == 0)

if (Soy Derecho)

arbol → padre → derecho = arbol → izquierda;

arbol → padre → izquierdo = arbol → izquierda;

~~arbol → izquierdo → padre = arbol → padre;~~

if (arbol → izquierdo != 0)

arbol → izq → padre = arbol → padre;

~~else~~  
Borrar (arbol → dato);  
free (arbol)

⊗ arbol → izq → padre = 0;

Perdón por la desprolijidad, pero creo que la idea a nivel (pseudocódigo) queda clara.



Borrar Subarbol (nodo \* arbol)

if (arbol != 0)

Borrar Subarbol (arbol → izq)

Borrar Subarbol (arbol → der)

arbol → Borrar (arbol → data)  
free (arbol)

## ASM

obs: El único caso donde cambia <sup>la raíz</sup> el padre es si nivel == 0 y en ese caso el nuevo padre la nueva raíz es el nodo de la izquierda.

Section .data:

```
% define offset_data 0
% 0 11 offset_izq 8
% 0 11 offset_padre 16
% 0 11 offset_data 24
% 0 11 offset_borrar 32
```

Section .text:

```
global borrar_nivel_conectado_izquierda: rdi * arbol
; es: nivel
```

```
push rbp
mov rbp, rsp
push r12
push r13
```

```
shl r9, 32
```

```
shr rsi, 32
```

```
mov r12, rdi
mov r13, rsi } preserva los parámetros, ya que el call puede
                pisarlos
```

```
mov rdi, [r12]; rdi ahora es * arbol
```

```
xor rdx, rdx
```

```
xor rcx, rcx
```

```
call buscar_nivel
```

```
cmp r13, 0
```

```
jne fin
```

```
mov r8, [r12]
```

```
mov r9, [r8 + offset_izq]
```

```
mov [r8], r9
```

```
mov rdi, r12
```

obs



```

* fin:  pop r13
        pop r12
        pop rbp
        ret

```

```

global borrarSubarbol ; rdi = arbol
push rbp
mov rbp, rsp
push r12
sub rsp, 8
mov r11, rdi
cmp rdi, 0
je .fin

```

```

mov rdi, [rdi + offset - itq]
call borrarSubarbol

```

```

mov rdi, r12
mov rdi, [rdi + offset - der]
call borrarSubarbol

```

```

mov rdi, [r12 + offset - data]
mov rcx, [r12 + offset - borrar]
call rcx
mov rdi, r12
call free

```

```

* fin:
add rsp, 8
pop r12
pop rbp
ret

```

```

global buscarNivel ; rdi = arbol
                        esi = nivel
                        edx = nivel Actual
                        rcx = son Deu

```

```

push rbp
mov rbp, rsp
push r12
push r13
push r14
push r15 ; almacenar

```

```

mov r12, rdi
mov r13, esi
mov r14, rcx
mov r15, rcx

```

```

shl rsi, 32
shr rsi, 32
shl rdi, 32
shr rdi, 32

```

```

cmp rdi, 0
je .fin

```



```
cmp rdi, rsi
jge .nivel_nivel
```

```
mov rdi, [rdi + offset - 129]
inc rdi
xor rcx, rcx
call Busca_nivel
```

```
mov rdi, [r12 + offset - der]
mov rsi, r13
mov rdx, r14
inc rdx
mov rcx, 1
call Busca_nivel
jmp borrar fin
```

• nivel\_nivel :

```
mov rdi, [r12 + offset - der]
call borrar_subrol
```

```
cmp r13, 0
jne .nivel_No_Cero
```

```
mov rdi, [r12 + offset - 129]
mov [rdi + offset - pedre], 0
jmp .borrar
```

• Nivel\_No\_Cero :

```
cmp r15, 0  
je .Soy_129
```

```
mov rdi, [r12 + offset - pedre]
mov r8, [r12 + offset - 129]
```

```
cmp r15, 0
je .Soy_129
```

```
mov [rdi + offset - der], r8
jmp .Actualizar_Pedre
```

• Soy\_129 :

```
mov [rdi + offset - 129], r8
```

• Actualizar\_Pedre :

```
mov rdi, [r12 + offset - 129]
cmp rdi, 0
je .borrar
```



mov r8, [r12 + offset - rdx]  
mov [rdi + offset - rdx], r8

border:

mov rdx, [r12 + offset - border]  
mov rdi, [r12 + offset - data]

callee

mov rdi, r12

callee

cin

pop r15

pop r14

pop r13

pop r12

pop rbp

ret



2) a) La estrategia a utilizar será cargar en memoria los primeros 5 valores del vector. Con una máscara vamos a invertirlos byte a byte para tenerlos en little-endian.

Luego, con otra máscara vamos a deshacerlos del 5° elemento, es decir que procesaremos de a 4 elementos. (Es decir 4 Dwords).

Una vez hecho eso, con un rrr de sumas horizontales deberíamos tener lo que queremos.

Section .data

align 16

invertir: DB 0x00, 0x01, 0x02, ..., 0x0F; en la primer posición pongo el último byte hasta que en el último pongo el primer

arrestar: DB 0xFF, 0x0B, 0x0A, 0x09, 0xFF, 0x0B, 0x07, 0x06, 0xFF, 0x05, 0x04, 0x03, 0xFF, 0x02, 0x01, 0x00

; byte más significativo de cada dword con 0, los otros 3B con el valor correspondiente a  $2^i$

invers: times 4 dd 0xFF FF FF FF

pi: times 4 dd 3.14

Section .text:

ConvertirAuno: times 4 dd 0x00 00 00 01

push rbp  
mov rbp, rsp

xor rcx, rcx  
mov rcx, 4  
xor r2x, r2x  
xor rdx, rdx

movdqa xmm1, [invertir]

movdqa xmm2, [arrestar]

pxor xmm7, xmm7

pxor xmm0, xmm0; se usz en b



• ciclo:

movdqa xmm0, [rdi]; xmm0 = 20 | 21 | 22 | 23 | 24 |

pushb xmm0, xmm0, xmm1; 24 | 23 | 22 | 21 | 20 |

pushb xmm0, xmm0, xmm1; 00 23 | 00 22 | 00 21 | 00 20 |

phadd xmm0, xmm7; 0... 0 | 20+23 | 20+21 |

phadd xmm0, xmm7; 20+23+20+21 |

movd edi, xmm0

add r2x, r8

lez rdi, [rdi+12]; proceso de a 4

loop.ciclo

Asamblea



pop rbp  
ret.

b) luego desde donde dice hasta 202...

mov dq xmm3, xmm0

cvtdq2pd xmm0, xmm- ; ahora a3 de son doubles

por xmm3, xmm10 ; okFFFF si a3 impr  
0x0000 sino

mov dq xmm4, xmm3

not xmm4 ; okFFFF si a3 es pr  
0x0000 sino

and xmm3, xmm1 ; okol si a3 es impr  
0x0000 sino

and xmm4, xmm1 ; p1 si a3 es pr  
0 sino

add pd xmm3, xmm4 ; 0x00001 si a3 es impr  
p1 sino

mul pd xmm0, xmm3 ;  $a_i \times p_i$  si a3 es impr  
a3 sino

hadd pd xmm0, xmm7  
hadd pd xmm0, xmm7 ;  $\sum_{i=0}^3 a_i$  en parte baja de xmm0

add sd xmm8, xmm0

leq rdi, [rdi+12]

loop -c1c6

xor xmm0, xmm0

movdqu xmm5, xmm0 ; retorno en xmm0

pop rbp  
ret

⊕ movdqa xmm10, [impru] ;  
movdqa xmm11, [p1] ;  
movdqa xmm12, [convertir A Wno] ;

; luego viene el ciclo.



3) a) Supongamos que tenemos nuestro código

Inst 0  
Inst 1  
⋮

por ejemplo  
Inst n+2 está en esa  
dirección

Inst n  
Call X

$0xA0A1 \leftarrow \text{Inst } n+2$

Inst n+3  
⋮

Al llamarse pasa esto

Inst 0

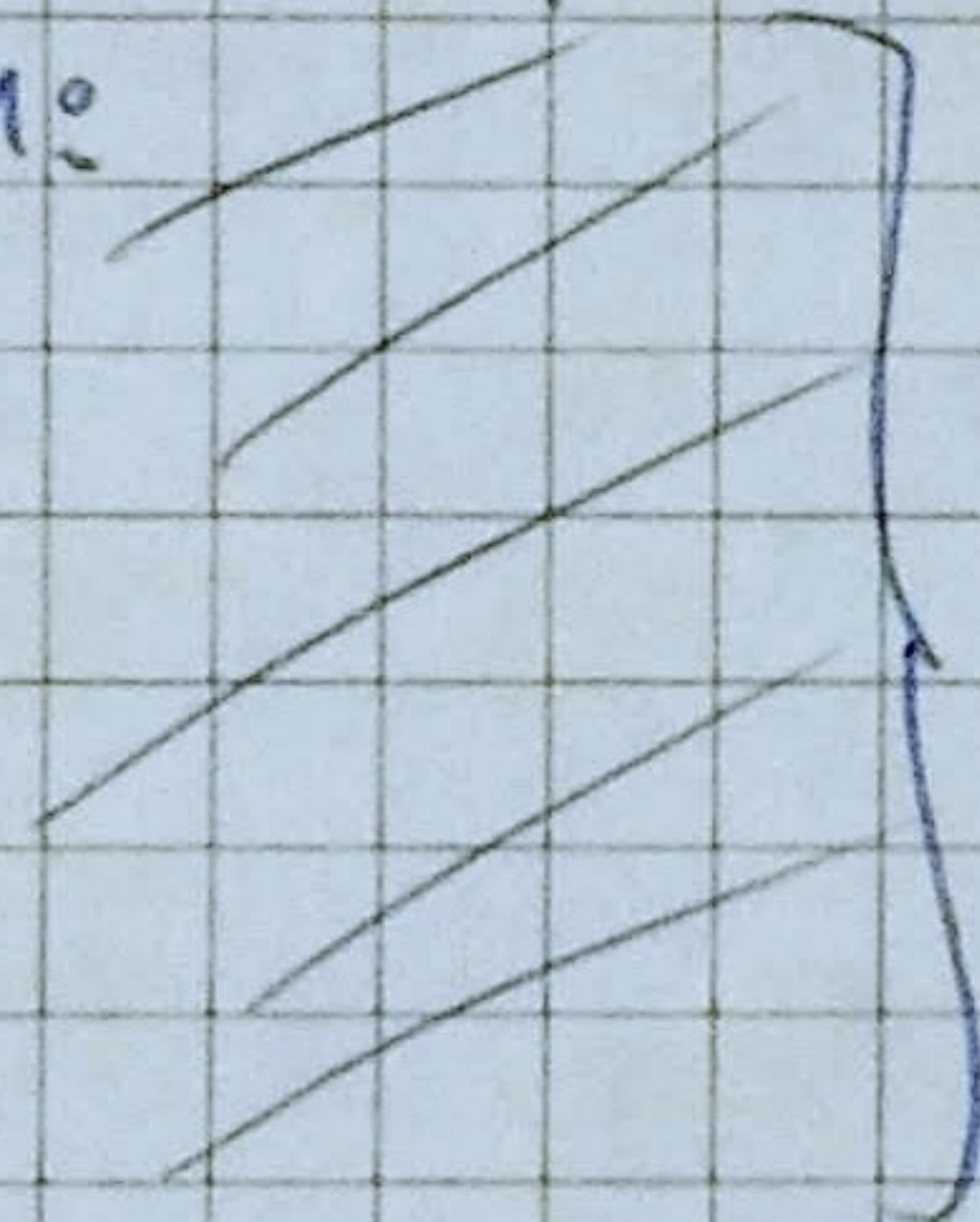
Inst 1

⋮

Inst n

Call X

$0xA0A1 \leftarrow$



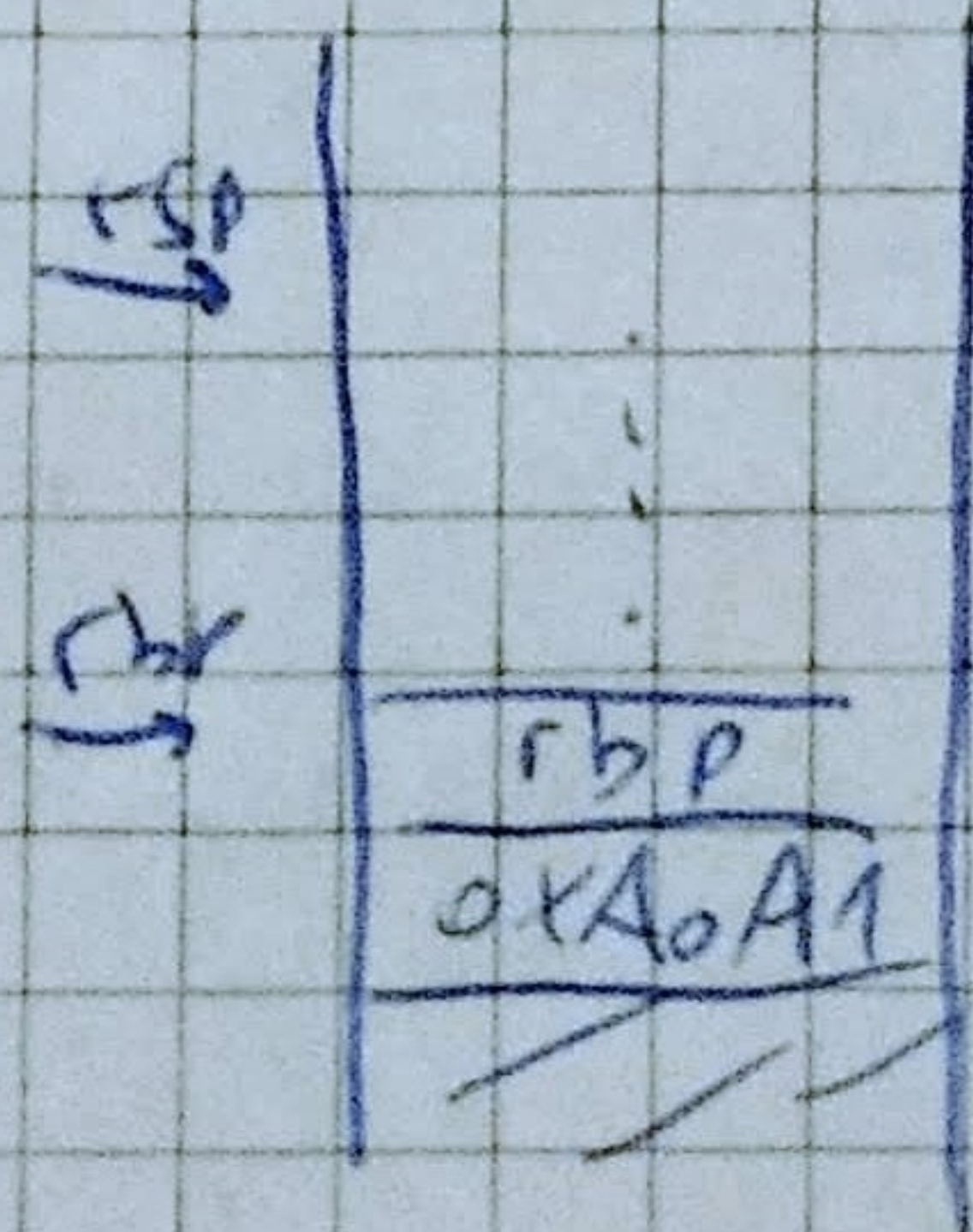
K bytes

$0xA0A1+K \leftarrow \text{Inst } n+2$   
Inst n+3

Cuando se llama a X,  $0xA0A1$  es la dirección de retorno, en su lugar hay que sumarle K.

Sabemos que se va a llamar a arreglar retorno luego de entrar al Stack Frame.

Entonces, en X la pila se ve así:





En  $[rbp+8]$  está el valor que queremos cambiar.

Arreglo retorno: ; rdi K

mov rdx, [rbp+8] ; el rbp es de x ; rsp apunta a la sig  
add rdx, rdi ; instrucción de x luego  
mov [rbp+8], rdx ; de esta función

ret

b) Desde la posición del ret de x que está en  $[rbp+8]$  durante K bytes,  
escribirá 0x90.

Modificando arreglo, reemplazando: ; rdi K

mov rdx, [rbp+8]

mov rcx, rdi

Ciclo:

mov [rdx], 0x90

inc rdx, [rdx+1]

loop. ciclo

ret.

c) Si el desensamblador "supiera" que luego de un call viene K bytes reemplazados,  
podría saltarlos fácilmente.

Como no lo sabe, el problema que se vea que puede traer algo que va  
a tener los K bytes y va a intentar convertirlos a instrucciones.

Eso puede derivar en que no le de ningún opcode válido y el desensamblador falle,

o bien que consiga traducir a instrucciones con operandos erróneos y el programa

desensamblado falle. O peor aún que genere instrucciones válidas que

afecten los datos y el normal funcionamiento del programa.

Estos problemas no los resuelven ni el pto. a ni el b y c que esos

"soluciones" en realidad son parches en tiempo de ejecución.

Por lo tanto, concludo que ~~no~~ la afirmación es falsa.